Implementation of a Shipboard Ballistic Missile Defense Processing Application Using the High Performance Embedded Computing Software Initiative (HPEC-SI) API

Joe Cook, Nathan Doss, Jane Kent, and ***Rick Pancoast***
Lockheed Martin, Maritime Systems and Sensors
Phone: 856-722-2354
Email Addresses: {joseph.j.cook, nathan.e.doss, jane.l.kent, rick.pancoast}@lmco.com

Jeremy Kepner
MIT Lincoln Laboratory
Phone: 781-981-3108
Email Address: kepner@ll.mit.edu

Abstract:

Advanced shipboard radar systems will be required to detect, track and classify ballistic missile targets and re-entry vehicles, as well as perform traditional Anti-Air Warfare (AAW) operations. New Open Architecture Systems, including COTS hardware as well as open system software, are required to implement the necessary algorithms for successful missile defense. Lockheed Martin MS2 has been developing such open architecture systems for the next generation Aegis Combat Systems, which will include the embedded equipment and computer programs that are necessary for an effective missile defense. Lockheed Martin has made extensive use of Open Architecture (OA) software and industry standard Application Programming Interfaces (APIs) in order to provide the Navy and Missile Defense Agency with efficient, open architecture software that exhibits unprecedented Portability across computing platforms, vendor design environments, processor architectures and technology upgrades.

As we move forward with development of a deployable shipboard missile defense system, it has become obvious that a state-of-the-art, C++ based object oriented design environment and signal processing API Library would be extremely beneficial in the development of open architecture application software, with the advantageous portability features provided by the original C-based VSIPL API. For this reason, Lockheed Martin has been an active participant in development of a next generation C++ signal and image processing API through the High Performance Embedded Computing Software Initiative (HPEC-SI).

This briefing describes an effort to implement advanced Shipboard Ballistic Missile Defense (SBMD) application algorithms utilizing HPEC-SI. Shipboard application code, previously written in the C programming language for conventional COTS PowerPC-based embedded architectures, is being converted by Lockheed Martin MS2, as an HPEC-SI Demonstration, to run under the HPEC-SI API. The C code, designed to run in a C environment, will be converted to the HPEC-SI API standard to run under a true C++ Object Oriented environment, and will eventually take advantage of the HPEC-SI parallel processing features.

Of particular interest in this conversion is a comparison of key DoD processing algorithms executed on a conventional, embedded processing architecture using C and C application libraries, as compared with execution in an embedded HPEC-SI processing environment The goals of this effort were to:

- Demonstrate a critical embedded DoD BMD signal processing application using the HPEC-SI API under development on the HPEC-SI initiative

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
01 FEB 2005	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Implementation of a Shipboard Ballistic Missile Defense Processing Application Using the High Performance Embedded Computing Software Initiative (HPEC-SI) API	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Lockheed Martin, Maritime Systems and Sensors; MIT Lincoln Laboratory**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM00001742, HPEC-7 Volume 1, Proceedings of the Eighth Annual High Performance Embedded Computing (HPEC) Workshops, 28-30 September 2004 Volume 1., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**37**	

- Compare the engineering development metrics, contrasting the conventional C API software development environment with the C++ HPEC-SI Object Oriented software development environment, and
- Compare relative code size and development cost with the C API

In this briefing, we describe the porting of several of the signal processing algorithms that have been developed using C-based VSIPL, and port them to the HPEC-SI VSIPL++ API under development. As part of this process, the HPEC-SI community will receive valuable feedback regarding the HPEC-SI API implementation, including the development process, development metrics, development environment issues and key library functions. Eventually, the open architecture HPEC-SI VSIPL++ code developed for the Navy and MDA will be ported to a tactical system for deployment on Aegis cruisers and destroyers.

LOCKHEED MARTIN

Abstract

Agenda

Implementation of a Shipboard Ballistic Missile Defense Processing Application Using the High Performance Embedded Computing Software Initiative (HPEC-SI) API

Jane Kent
Joseph Cook
Rick Pancoast
Nathan Doss
Jordan Lusterman

Lockheed Martin
Maritime Systems & Sensors (MS2)

HPEC 2004
30 Sep 2004

Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion

Overview

LOCKHEED MARTIN

HPEC Software Initiative (HPEC-SI) Goals

- Develop software technologies for embedded parallel systems to address:
  - Portability
  - Productivity
  - Performance
- Deliver quantifiable benefits

Current HPEC-SI Focus

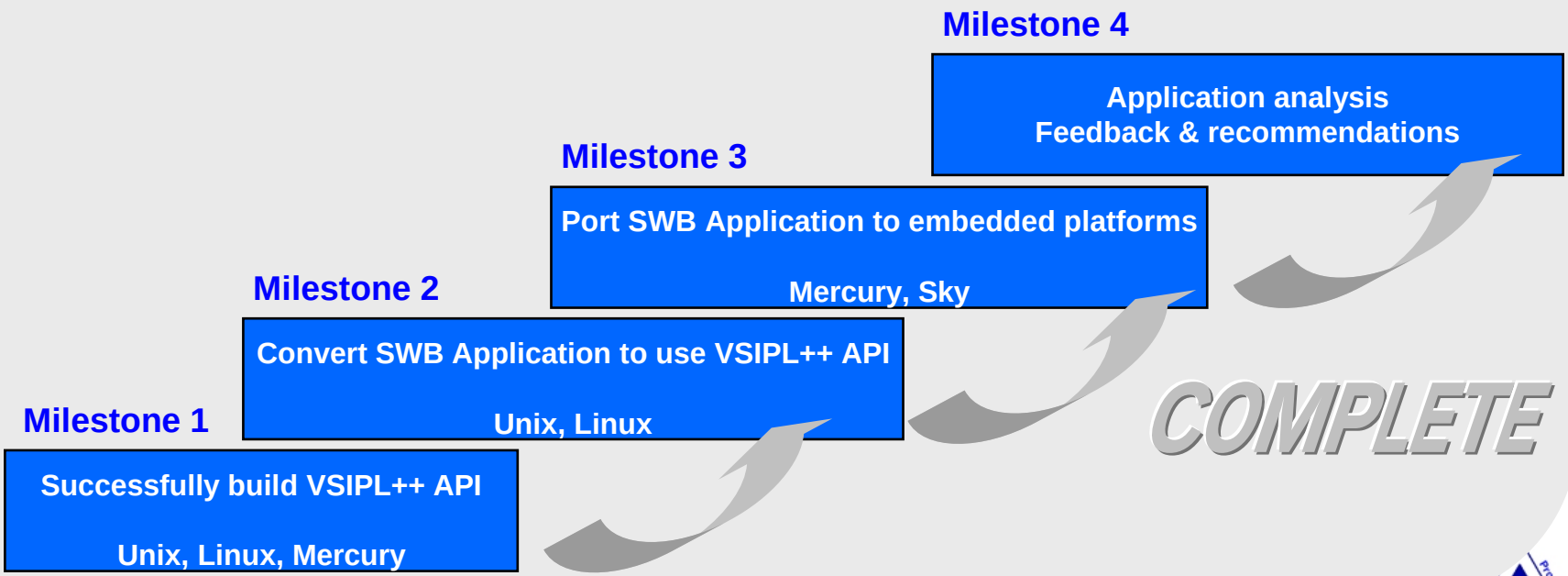
- Development of the VSIPL++ and Parallel VSIPL++ Standards
- VSIPL++
  - A C++ API based on concepts from VSIPL (an existing, industry accepted standard for signal processing)
  - VSIPL++ allows us to take advantage of useful C++ features
- Parallel VSIPL++ is an extension to VSIPL++ for multi-processor execution

VSIPL++ Development Process

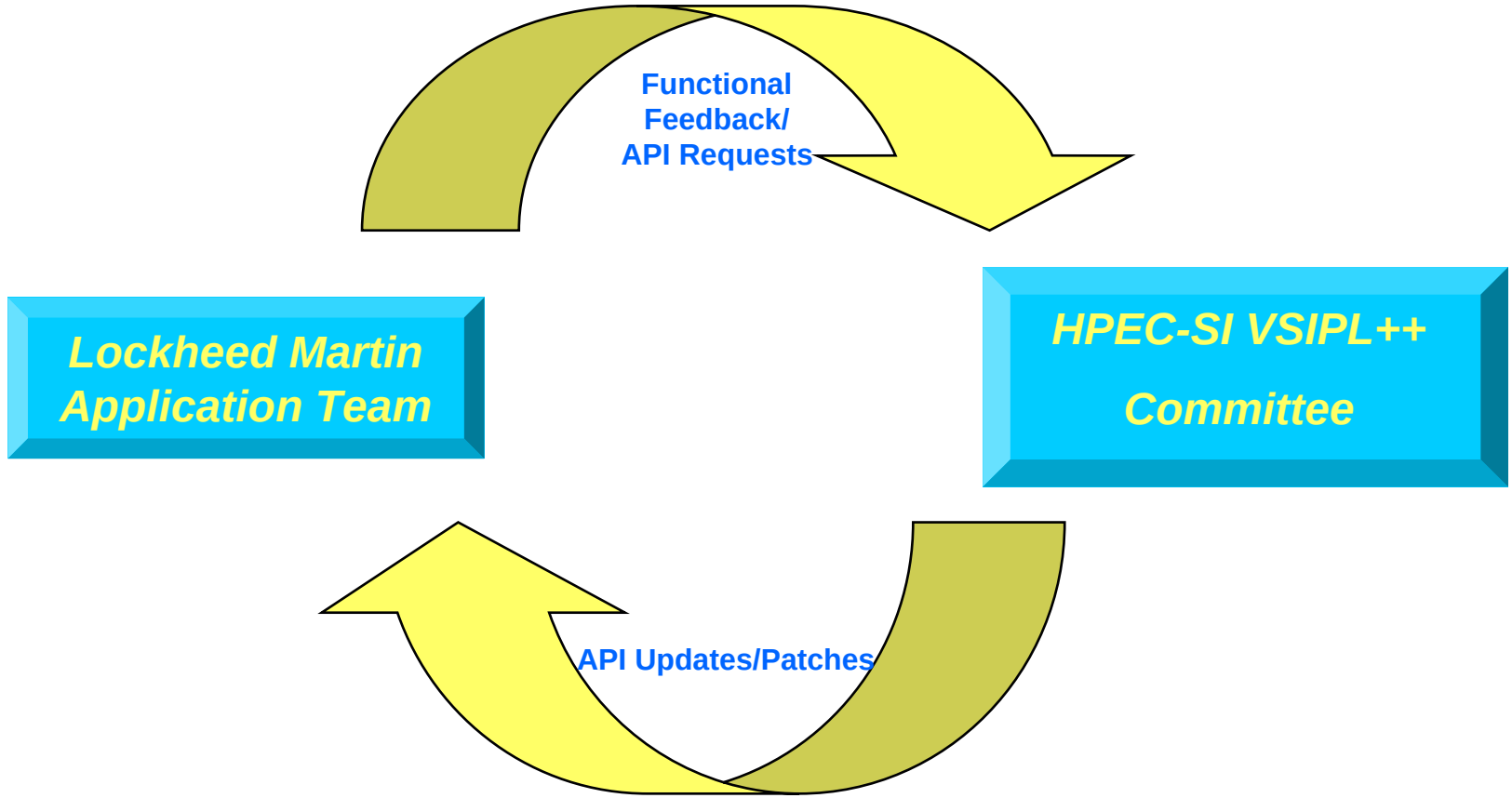
- Development of the VSIPL++ Reference Specification
- Creation of a reference implementation of VSIPL++
- Creation of demo applications

Lockheed Martin Demonstration Goals

- Use CodeSourcery's VSIPL++ reference implementation in a main-stream DoD Digital Signal Processor Application
- Utilize existing "real-world" tactical application Synthetic WideBand (SWB) Radar Mode. The original code was developed for the United States Navy and MDA under contract for improved S-Band Discrimination. SWB is continuing to be evolved by MDA for Aegis BMD signal processor.
- Identify areas for improved or expanded functionality and usability



VSIPL++
Standards - Development Loop



During development, there was a continuous loop of change requests/feedback, and API updates and patches

Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion

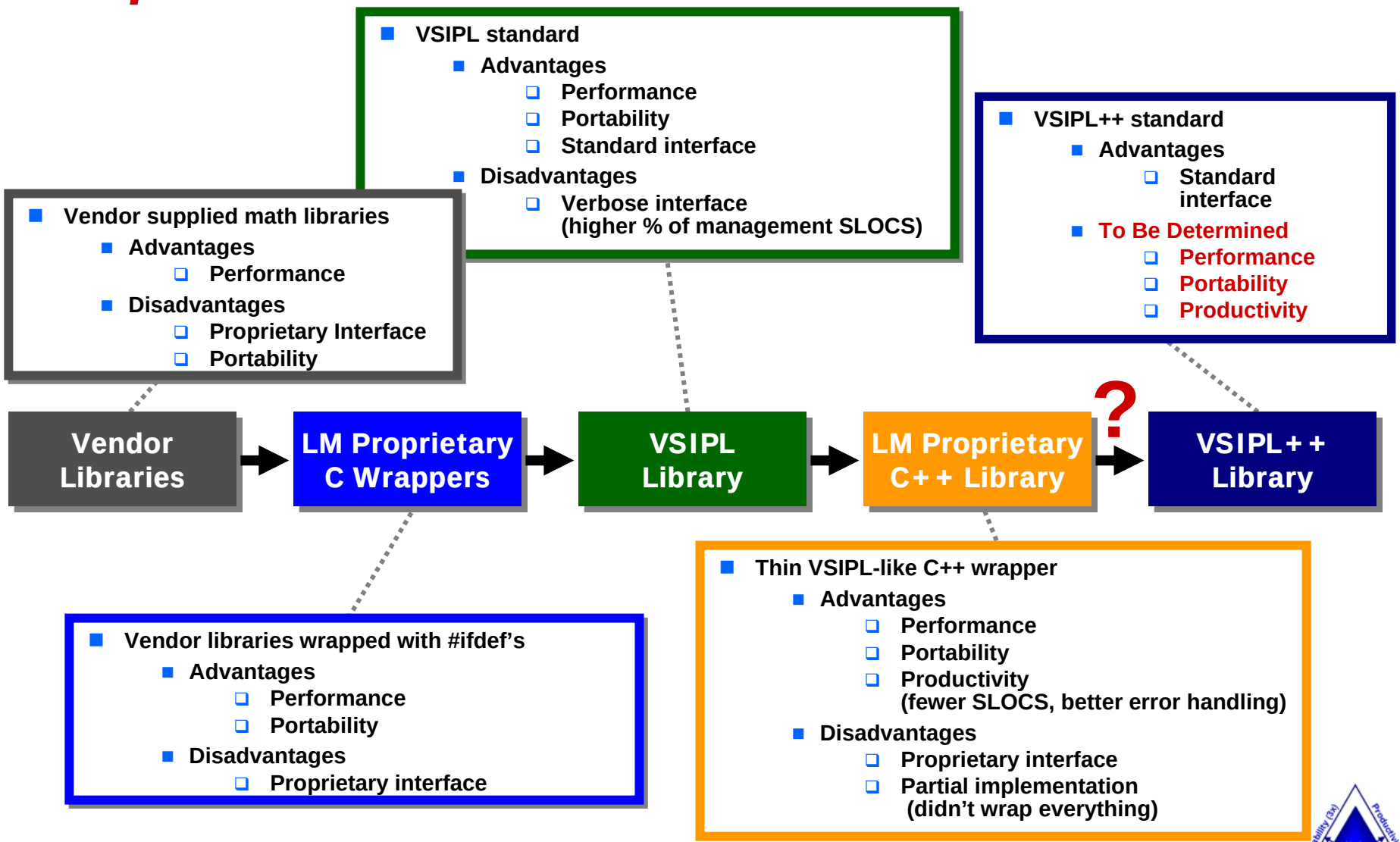
Lockheed Martin Software Risk Reduction Issues

- General mission system requirements
  - Maximum use of COTS equipment, software and commercial standards
  - Support high degree of software portability and vendor interoperability
- Software Risk Issues
  - Real-time operation
    - Latency
    - Bandwidth
    - Throughput
  - Portability and re-use
    - Across architectures
    - Across vendors
    - With vendor upgrades
  - Real-time signal processor control
    - System initialization
    - Fault detection and isolation
    - Redundancy and reconfiguration
  - Scalability to full tactical signal processor

Lockheed Martin Software Risk Reduction Efforts

- Benchmarks on vendor systems (CSPI, Mercury, HP, Cray, Sky, etc.)
  - Communication latency/throughput
  - Signal processing functions (e.g., FFTs)
  - Applications
- Use of and monitoring of industry standards
  - Communication standards: MPI, MPI-2, MPI/RT, Data Re-org, CORBA
  - Signal processing standards: VSIPL, VSIPL++
- Technology refresh experience with operating system, network, and processor upgrades (e.g., CSPI, SKY, Mercury)
- Experience with VSIPL
  - Participation in standardization effort
  - Implementation experience
    - Porting of VSIPL reference implementation to embedded systems
    - C++ wrappers
  - Application modes developed
    - Programmable Energy Search
    - Programmable Energy Track
    - Cancellation
    - Moving Target Indicator
    - Pulse Doppler
    - Synthetic Wideband

Lockheed Martin Math Library Experience



Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion

Application Overview

LOCKHEED MARTIN

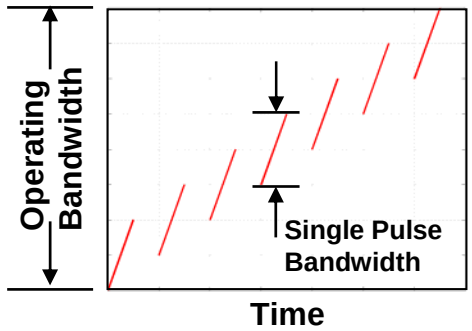
- The Lockheed Martin team took existing Synthetic Wideband application, developed and targeted for Aegis BMD signal processor implementation, and rewrote it to use and take advantage of the VSIPL++
- The SWB Application achieves a high bandwidth resolution using narrow bandwidth equipment, for the purposes of extracting target discriminant information from the processed range doppler image
- Synthetic Wideband was chosen because:
  - It exercises a number of algorithms and operations commonly used in our embedded signal processing applications
  - Its scope is small enough to finish the task completely, yet provide meaningful feedback in a timely manner
  - Main-stream DoD application

Application Overview – Synthetic WideBand Processing

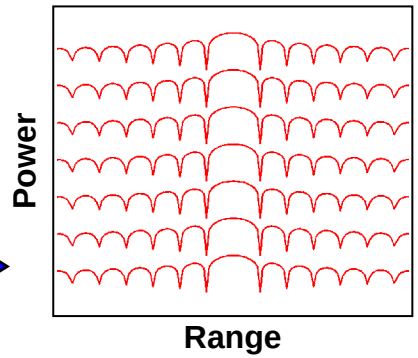
LOCKHEED MARTIN

By using "Stepped" medium band pulses, and specialized algorithms,
an effective "synthetic" wide band measurement can be obtained

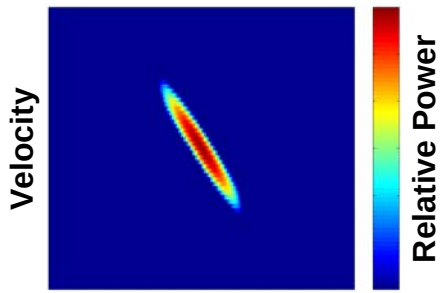
Synthetic Wideband Waveform Processing



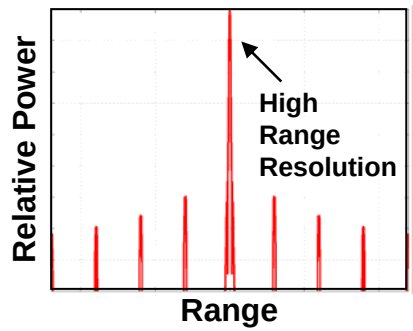
1. *Transmit and Receive Mediumband Pulses*



2. *Pulse Compress Mediumband Pulses*

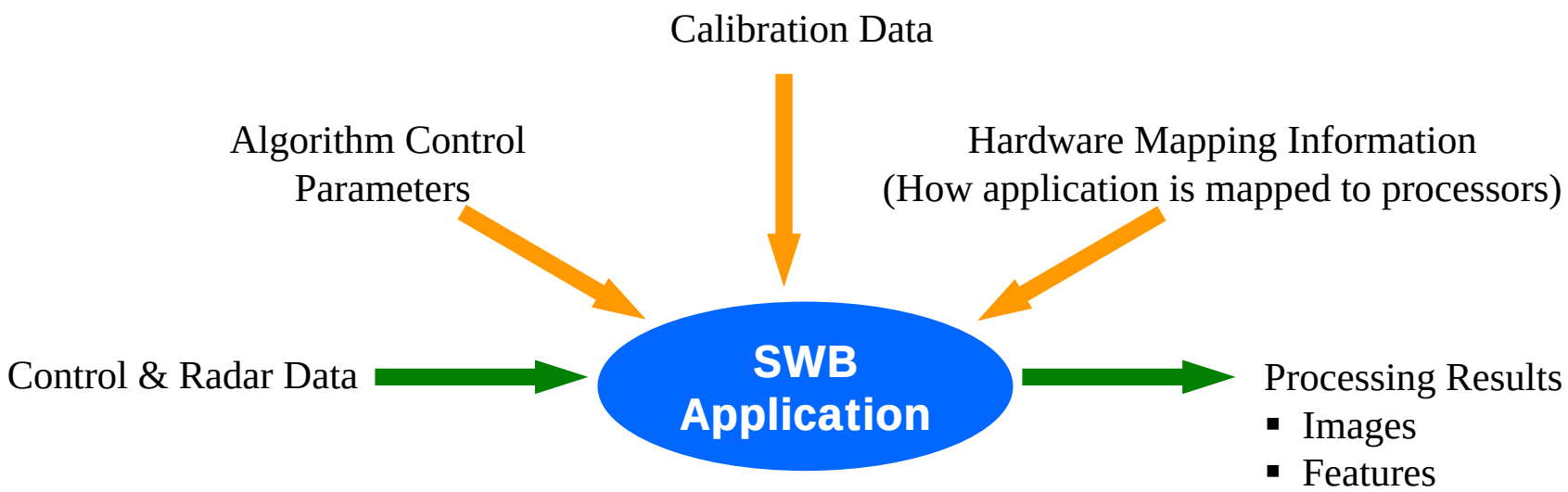


3. *Coherently Combine Mediumband Pulses to Obtain Synthetic Wideband Response*

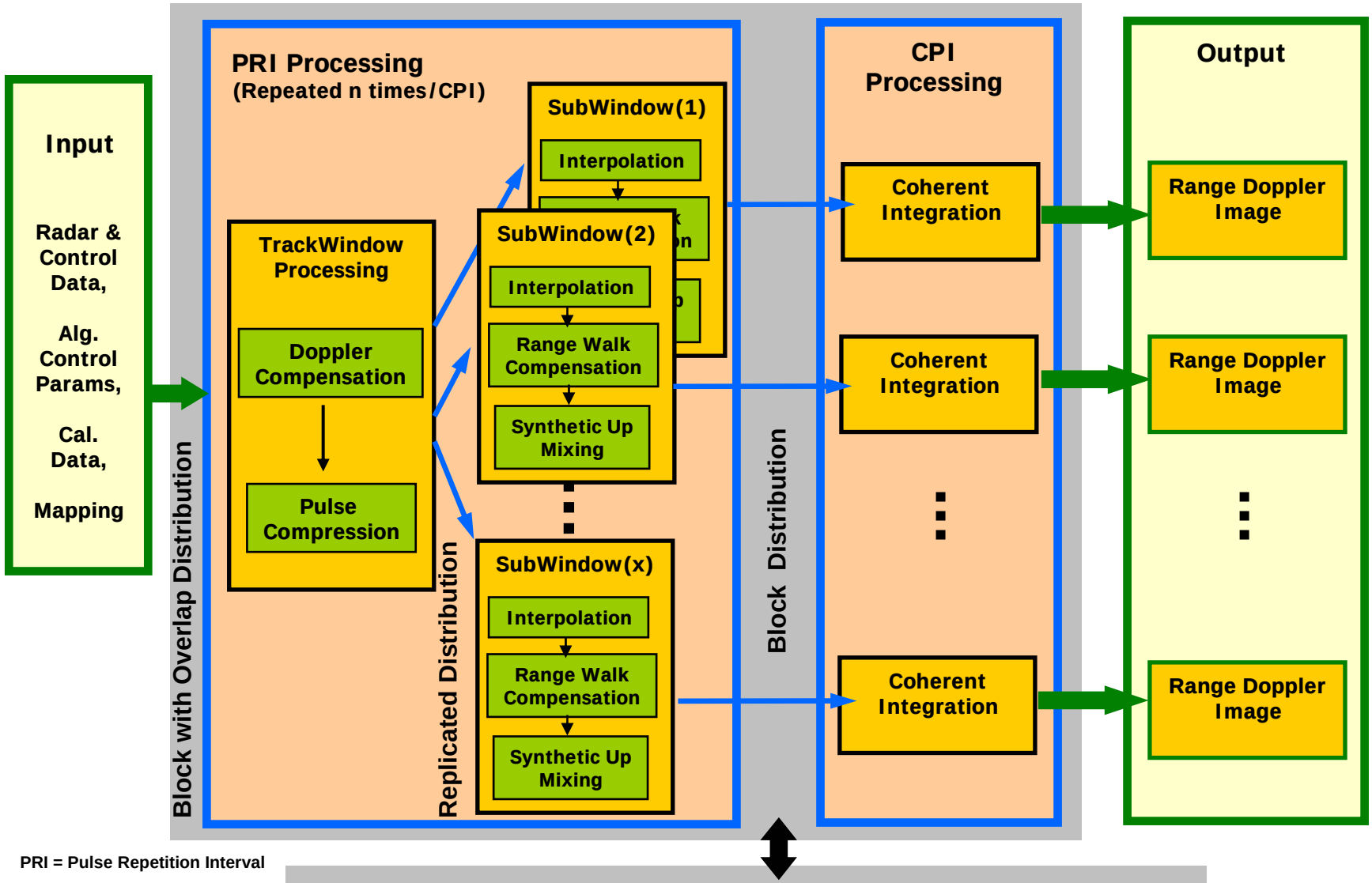


- *Requires accurate knowledge of target motion over waveform duration*
- *Requires phase calibration as a function of mediumband pulse center frequency*

Application Interface



Processing Flow



PRI = Pulse Repetition Interval

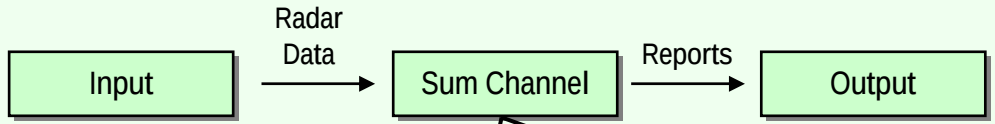
CPI = Coherent Pulse Interval

Software Architecture

LOCKHEED MARTIN

Application "main"

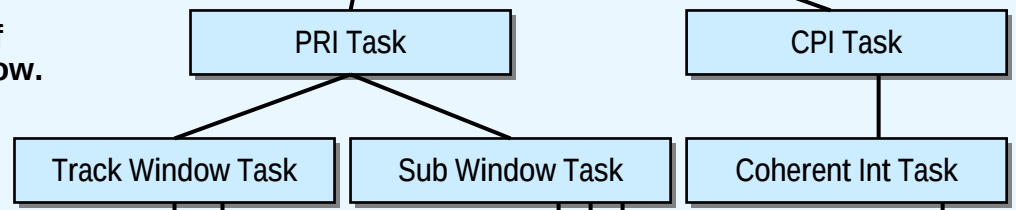
Ties together a set of tasks to build the overall application



Tasks

Data-parallel code that can be mapped to a set of processors and/or strung together into a data flow. Tasks are responsible for:

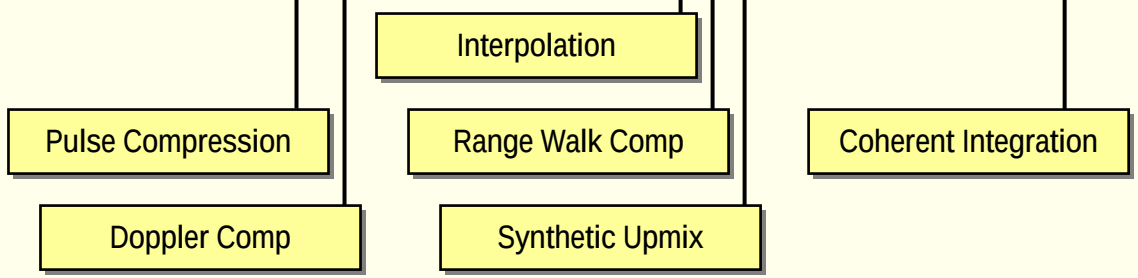
- **Sending and/or receiving data**
- **Processing the data (using the algorithms)**
- **Reading the stimulus control data and passing any needed control parameters into the algorithms**



Algorithms

Library of higher-level, application-oriented math functions with VSIPL-like interface

- **Interface uses views for input/output**
- **Algorithms never deal explicitly with data distribution issues**



HPEC-SI development involved modification of the algorithms

Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion

Algorithm Case Study Overview

LOCKHEED MARTIN

- Goal
  - Show how we reached some of our VSIPL++ conclusions by walking through the series of steps needed to convert a part of our application from VSIPL to VSIPL++
- Algorithm
  - Starting point
    - Simplified version of a pulse compression kernel
    - Math: output = ifft(fft(input) * reference)
  - Add requirements
    - Error handling
    - Decimate input
    - Support both single and double precision
    - Port application to embedded system

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

VSIPL

```
void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in);

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);

  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}
```

Observations

- **VSIPL++ code has fewer SLOCS than VSIPL code (5 VSIPL++ SLOCS vs. 13 VSIPL SLOCS)**
- **VSIPL++ syntax is more complex than VSIPL syntax**
  - **Syntax for FFT object creation**
  - **Extra set of parenthesis needed in defining Domain argument for FFT objects**
- **VSIPL code includes more management SLOCS**
  - **VSIPL code must explicitly manage temporaries**
  - **Must remember to free temporary objects and FFT operators in VSIPL code**
- **VSIPL++ code expresses core algorithm in fewer SLOCS**
  - **VSIPL++ code expresses algorithm in one line, VSIPL code in three lines**
  - **Performance of VSIPL++ code may be better than VSIPL code**

VSIPL++

```
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref,
                   const vsip::Vector< std::complex<float> > &out) {
  int size = in.size();

  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft(in), out );
}
```

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement **Catch any errors and propagate error status**

VSIPL

```
int pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  int valid = 0;
  vsip_length size = vsip_cvgetlength_f(in);

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  if (forwardFft && inverseFft && tmpView1 && tmpView2)  {
    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    valid=1;
  }

  if (tmpView1)  vsip_cvalldestroy_f(tmpView1);
  if (tmpView2)  vsip_cvalldestroy_f(tmpView2);
  if (forwardFft) vsip_fft_destroy_f(forwardFft);
  if (inverseFft) vsip_fft_destroy_f(inverseFft);
  return valid;
}
```

VSIPL++

```
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref,
                   const vsip::Vector< std::complex<float> > &out) {
  int size = in.size();

  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD>  forwardFft ((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE>  inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft(in), out );
}
```

Observations

- VSIPL code additions are highlighted
  - No changes to VSIPL++ function due to VSIPL++ support for C++ exceptions
  - 5 VSIPL++ SLOCS vs. 17 VSIPL SLOCS
- VSIPL behavior not defined by specification if there are errors in fft and vector multiplication calls
  - For example, if lengths of vector arguments unequal, implementation may core dump, stop with error message, silently write past end of vector memory, etc
  - FFT and vector multiplication calls do not return error codes

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement **Decimate input by N prior to first FFT**

VSIPL

```
void pulseCompress( int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  vsip_cvputstride_f(in, decimationFactor);
  vsip_cvputlength_f(in, size);

  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);

  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}
```

VSIPL++

```
void pulseCompress(int decimationFactor,  const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref
                   const vsip::Vector< std::complex<float> > &out) {
  int size = in.size() / decimationFactor;
  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```

Observations

- SLOC count doesn't change all that much for VSIPL or VSIPL++ code
  - 2 changed line for VSIPL
  - 3 changed lines for VSIPL++
  - 2 additional SLOCS for VSIPL
  - 1 additional SLOC for VSIPL++
- VSIPL version of code has a side-effect
  - The input vector was modified and not restored to original state
  - This type of side-effect was the cause of many problems/bugs when we first started working with VSIPL

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement **Decimate input by N prior to first FFT, no side-effects**

VSIPL

```
void pulseCompress( int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length savedSize   = vsip_cvgetlength_f(in);
  vsip_length savedStride = vsip_cvgetstride_f(in);
  vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvputlength_f(in, size);
  vsip_cvputstride_f(in, decimationFactor);
  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);
  vsip_cvputlength_f(in, savedSize);
  vsip_cvputstride_f(in, savedStride);

  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}
```

Observations

- **VSIPL code must save away the input vector state prior to use and restore it before returning**
- **Code size changes**
  - **VSIPL code requires 4 additional SLOCS**
  - **VSIPL++ code does not change from prior version**

VSIPL++

```
void pulseCompress(int decimationFactor,  const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref
                   const vsip::Vector< std::complex<float> > &out) {
  int size = in.size() / decimationFactor;
  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement **Support both single and double precision floating point**

VSIPL

Single Precision

```
void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in);

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_f *tmpVie
  vsip_cvview_f *tmpVie

  vsip_ccfftop_f(forwardI
  vsip_cvmul_f(tmpView
  vsip_ccfftop_f(inverseF

  vsip_cvalldestroy_f(tm
  vsip_cvalldestroy_f(tm
  vsip_fft_destroy_
  vsip_fft_destroy_
}
```

Double Precision

```
void pulseCompress(vsip_cvview_d *in, vsip_cvview_d *ref, vsip_cvview_d *out) {
  vsip_length size = vsip_cvgetlength_d(in);

  vsip_fft_d *forwardFft = vsip_ccfftop_create_d(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_d *inverseFft = vsip_ccfftop_create_d(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_d *tmpView1 = vsip_cvcreate_d(size, VSIP_MEM_NONE);
  vsip_cvview_d *tmpView2 = vsip_cvcreate_d(size, VSIP_MEM_NONE);

  vsip_ccfftop_d(forwardFft, in, tmpView1);
  vsip_cvmul_d(tmpView1, ref, tmpView2);
  vsip_ccfftop_d(inverseFft, tmpView2, out);

  vsip_cvalldestroy_d(tmpView1);
  vsip_cvalldestroy_d(tmpView2);
  vsip_fft_destroy_d(forwardFft);
  vsip_fft_destroy_d(inverseFft);
}
```

Observations

- **VSIPL++ code has same SLOC count as original**
  - **Uses c++ templates (3 lines changed)**
  - **Syntax is slightly more complicated**
- **VSIPL code doubles in size**
  - **Function must first be duplicated**
  - **Small changes must then be made to code (*i.e.*, changing _f to _d)**

VSIPL++

```
template<class T, class U, class V> void pulseCompress(const T &in, const U &ref, const V &out) {
  int size = in.size();

  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft(in), out );
}
```

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement **Support all previously stated requirements**

VSIPL

Single Precision

```
void pulseCompress(int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length savedSize   = vsip_cvgetlength_f
  vsip_length savedStride = vsip_cvgetstride_f(

  vsip_length size = vsip_cvgetlength_f(in) / de

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(s
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(s

  if (forwardFft && inverseFft && tmpView1 &&
  {
    vsip_cvputlength_f(in, size);
    vsip_cvputstride_f(in, decimationFactor);

    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2
    vsip_ccfftop_f(inverseFft, tmpView2, ou

    vsip_cvputlength_f(in, savedSize);
    vsip_cvputstride_f(in, savedStride);
  }

  if (tmpView1)   vsip_cvalldestroy_f(tmpV
  if (tmpView2)   vsip_cvalldestroy_f(tmpV
  if (forwardFft) vsip_fft_destroy_f(forwardF
  if (inverseFft) vsip_fft_destroy_f(inverseF
}
```

Double Precision

```
void pulseCompress(int decimationFactor, vsip_cvview_d *in, vsip_cvview_d *ref, vsip_cvview_d *out) {
  vsip_length savedSize   = vsip_cvgetlength_d(in);
  vsip_length savedStride = vsip_cvgetstride_d(in);

  vsip_length size = vsip_cvgetlength_d(in) / decimationFactor;

  vsip_fft_d *forwardFft = vsip_ccfftop_create_d(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_d *inverseFft = vsip_ccfftop_create_d(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_d *tmpView1 = vsip_cvcreate_d(size, VSIP_MEM_NONE);
  vsip_cvview_d *tmpView2 = vsip_cvcreate_d(size, VSIP_MEM_NONE);

  if (forwardFft && inverseFft && tmpView1 && tmpView2)
  {
    vsip_cvputlength_d(in, size);
    vsip_cvputstride_d(in, decimationFactor);

    vsip_ccfftop_d(forwardFft, in, tmpView1);
    vsip_cvmul_d(tmpView1, ref, tmpView2);
    vsip_ccfftop_d(inverseFft, tmpView2, out);

    vsip_cvputlength_d(in, savedSize);
    vsip_cvputstride_d(in, savedStride);
  }

  if (tmpView1)   vsip_cvalldestroy_d(tmpView1);
  if (tmpView2)   vsip_cvalldestroy_d(tmpView2);
  if (forwardFft) vsip_fft_destroy_d(forwardFft);
  if (inverseFft) vsip_fft_destroy_d(inverseFft);
}
```

Observations

- **Final SLOC count**
  - **VSIPL++ -- 6 SLOCS**
  - **VSIPL -- 40 SLOCS (20 each for double and single precision versions)**

VSIPL++

```
template<class T, class U, class V> void pulseCompress(int decimationFactor, const T &in, const U &ref, const V &out) {
  int size = in.size() / decimationFactor;

  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```


HPEC SI — Productivity (3x) — Performance (1.5x)

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement **Port application to high performance embedded systems**

VSIPL

Single Precision

```
void pulseCompress(int decimationFact
  vsip_length savedSize   = vsip_cvgetle
  vsip_length savedStride = vsip_cvgets

  vsip_length size = vsip_cvgetlength_f(i

  vsip_fft_f *forwardFft = vsip_ccfftop_cr
  vsip_fft_f *inverseFft = vsip_ccfftop_cre

  vsip_cvview_f *tmpView1 = vsip_cvcre
  vsip_cvview_f *tmpView2 = vsip_cvcre

  if (forwardFft && inverseFft && tmpVie
  {
    vsip_cvputlength_f(in, size);
    vsip_cvputstride_f(in, decimationFact

    vsip_ccfftop_f(forwardFft, in, tmpView
    vsip_cvmul_f(tmpView1, ref, tmpView
    vsip_ccfftop_f(inverseFft, tmpView2, 

    vsip_cvputlength_f(in, savedSize);
    vsip_cvputstride_f(in, savedStride);
  }

  if (tmpView1)   vsip_cvalldestroy_f(tmp
  if (tmpView2)   vsip_cvalldestroy_f(tmp
  if (forwardFft) vsip_fft_destroy_f(forwar
  if (inverseFft) vsip_fft_destroy_f(invers
}
```

Observations

- **Port to embedded Mercury system**
  - **Hardware: Mercury VME chassis with PowerPC compute nodes**
  - **Software: Mercury beta release of MCOE 6.0 with linux operating system. Mercury provided us with instructions for using GNU g++ compiler**
  - **No lines of application code had to be changed**
- **Port to embedded Sky system**
  - **Hardware: Sky VME chasis with PowerPC compute nodes**
  - **Software: Sky provided us with a modified version of their standard compiler (added a GNU g++ based front-end)**
  - **No lines of application code had to be changed**
- **Future availability of C++ with support for C++ standard**
  - **Improved C++ support is in Sky and Mercury product roadmaps**
  - **Support for C++ standard appears to be improving industry wide**

VSIPL++

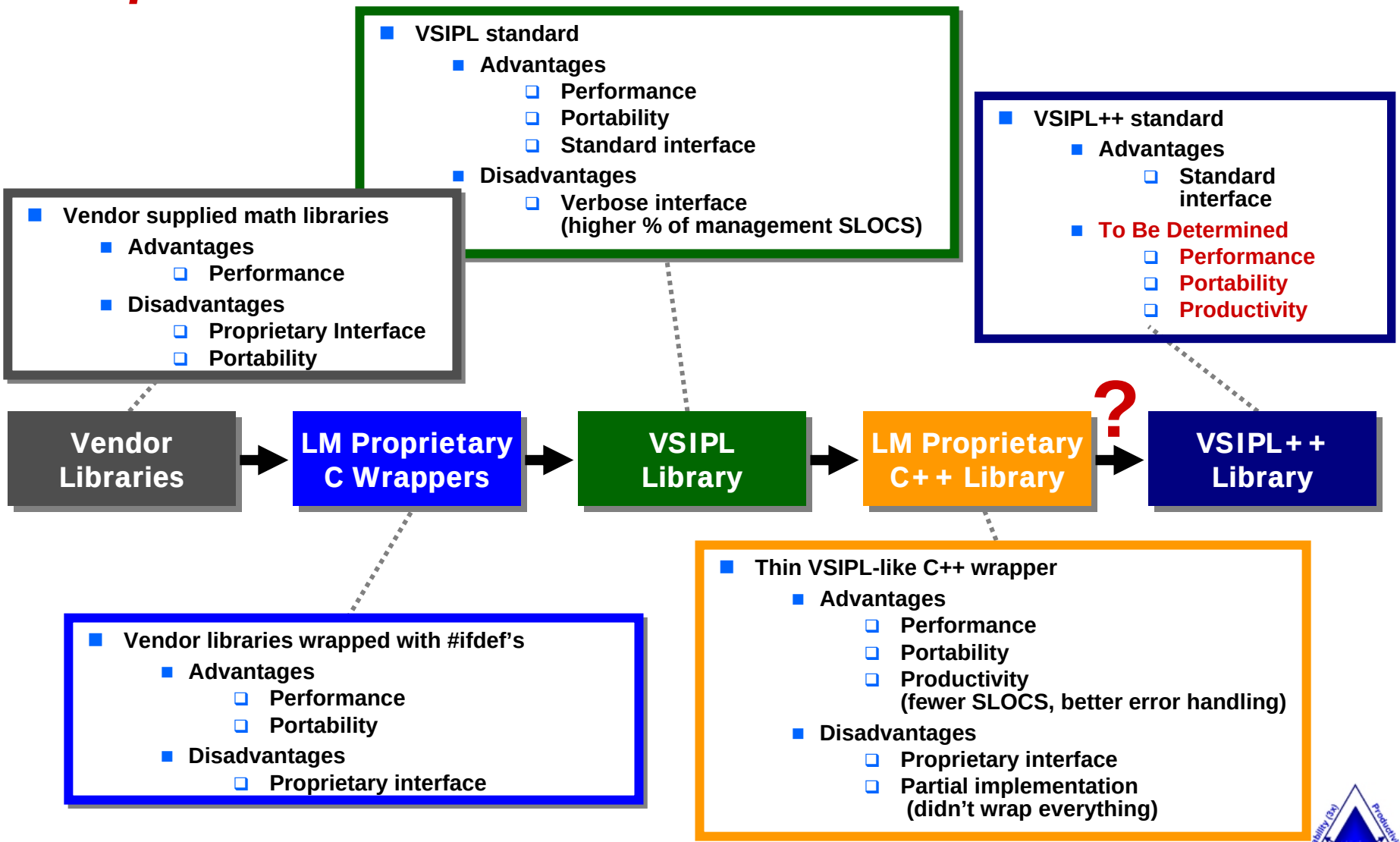
```
template<class T, class U, class V> void pulseCompress(int decimationFactor, const T &in, const U &ref, const V &out) {
  int size = in.size() / decimationFactor;

  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```


HPEC SI Productivity (3x) Performance (1.5x)

Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion

Lockheed Martin Math Library Experience



Conclusion

- Standard interface
- Productivity
  - A VSIPL++ user's guide, including a set of examples would have been helpful
  - The learning curve for VSIPL++ can be somewhat steep initially
  - Fewer lines of code are needed to express mathematical algorithms in VSIPL++
  - Fewer maintenance SLOCS are required for VSIPL++ programs
- Portability
  - VSIPL++ is portable to platforms with support for standard C++
  - Most vendors have plans to support advanced C++ features required by VSIPL++
- Performance
  - VSIPL++ provides greater opportunity for performance
  - Performance-oriented implementation is not currently available to verify performance

Lockheed Martin goals are well aligned with VSIPL++ goals

UNANIMATED BACKUPS

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

VSIPL

```
void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in);

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);

  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}
```

Observations

- **VSIPL++ code has fewer SLOCS than VSIPL code (5 VSIPL++ SLOCS vs. 13 VSIPL SLOCS)**
- **VSIPL++ syntax is more complex than VSIPL syntax**
  - **Syntax for FFT object creation**
  - **Extra set of parenthesis needed in defining Domain argument for FFT objects**
- **VSIPL code includes more management SLOCS**
  - **VSIPL code must explicitly manage temporaries**
  - **Must remember to free temporary objects and FFT operators in VSIPL code**
- **VSIPL++ code expresses core algorithm in fewer SLOCS**
  - **VSIPL++ code expresses algorithm in one line, VSIPL code in three lines**
  - **Performance of VSIPL++ code may be better than VSIPL code**

VSIPL++

```
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref,
                   const vsip::Vector< std::complex<float> > &out) {
  int size = in.size();

  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft(in), out );
}
```

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement Catch any errors and propagate error status

VSIPL

```
int pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  int valid = 0;
  vsip_length size = vsip_cvgetlength_f(in);

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  if (forwardFft && inverseFft && tmpView1 && tmpView2)  {
    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    valid=1;
  }

  if (tmpView1)  vsip_cvalldestroy_f(tmpView1);
  if (tmpView2)  vsip_cvalldestroy_f(tmpView2);
  if (forwardFft) vsip_fft_destroy_f(forwardFft);
  if (inverseFft) vsip_fft_destroy_f(inverseFft);
  return valid;
}
```

VSIPL++

```
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref,
                   const vsip::Vector< std::complex<float> > &out) {
  int size = in.size();

  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD>  forwardFft ((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE>  inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft(in), out );
}
```

Observations

- **VSIPL code additions are highlighted**
  - **No changes to VSIPL++ function due to VSIPL++ support for C++ exceptions**
  - **5 VSIPL++ SLOCS vs. 17 VSIPL SLOCS**
- **VSIPL behavior not defined by specification if there are errors in fft and vector multiplication calls**
  - **For example, if lengths of vector arguments unequal, implementation may core dump, stop with error message, silently write past end of vector memory, etc**
  - **FFT and vector multiplication calls do not return error codes**

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement Decimate input by N prior to first FFT

VSIPL

```
void pulseCompress( int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

  vsip_cvputstride_f(in, decimationFactor);
  vsip_cvputlength_f(in, size);

  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);

  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}
```

VSIPL++

```
void pulseCompress(int decimationFactor,  const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref
                   const vsip::Vector< std::complex<float> > &out) {
  int size = in.size() / decimationFactor;
  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```

Observations

- **SLOC count doesn't change all that much for VSIPL or VSIPL++ code**
  - **2 changed line for VSIPL**
  - **3 changed lines for VSIPL++**
  - **2 additional SLOCS for VSIPL**
  - **1 additional SLOC for VSIPL++**
- **VSIPL version of code has a side-effect**
  - **The input vector was modified and not restored to original state**
  - **This type of side-effect was the cause of many problems/bugs when we first started working with VSIPL**

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement Decimate input by N prior to first FFT, no side-effects

VSIPL

```
void pulseCompress( int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length savedSize   = vsip_cvgetlength_f(in);
  vsip_length savedStride = vsip_cvgetstride_f(in);
  vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvputlength_f(in, size);
  vsip_cvputstride_f(in, decimationFactor);
  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);
  vsip_cvputlength_f(in, savedSize);
  vsip_cvputstride_f(in, savedStride);

  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}
```

Observations

- **VSIPL code must save away the input vector state prior to use and restore it before returning**
- **Code size changes**
  - **VSIPL code requires 4 additional SLOCS**
  - **VSIPL++ code does not change from prior version**

VSIPL++

```
void pulseCompress(int decimationFactor,  const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref
                   const vsip::Vector< std::complex<float> > &out) {
  int size = in.size() / decimationFactor;
  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement Support both single and double precision floating point

VSIPL

Single Precision

```
void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in);

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_f *tmpVie
  vsip_cvview_f *tmpVie

  vsip_ccfftop_f(forwardI
  vsip_cvmul_f(tmpView
  vsip_ccfftop_f(inverseF

  vsip_cvalldestroy_f(tm
  vsip_cvalldestroy_f(tm
  vsip_fft_destroy_
  vsip_fft_destroy_
}
```

Double Precision

```
void pulseCompress(vsip_cvview_d *in, vsip_cvview_d *ref, vsip_cvview_d *out) {
  vsip_length size = vsip_cvgetlength_d(in);

  vsip_fft_d *forwardFft = vsip_ccfftop_create_d(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_d *inverseFft = vsip_ccfftop_create_d(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_d *tmpView1 = vsip_cvcreate_d(size, VSIP_MEM_NONE);
  vsip_cvview_d *tmpView2 = vsip_cvcreate_d(size, VSIP_MEM_NONE);

  vsip_ccfftop_d(forwardFft, in, tmpView1);
  vsip_cvmul_d(tmpView1, ref, tmpView2);
  vsip_ccfftop_d(inverseFft, tmpView2, out);

  vsip_cvalldestroy_d(tmpView1);
  vsip_cvalldestroy_d(tmpView2);
  vsip_fft_destroy_d(forwardFft);
  vsip_fft_destroy_d(inverseFft);
}
```

Observations

- **VSIPL++ code has same SLOC count as original**
  - **Uses c++ templates (3 lines changed)**
  - **Syntax is slightly more complicated**
- **VSIPL code doubles in size**
  - **Function must first be duplicated**
  - **Small changes must then be made to code (*i.e.*, changing _f to _d)**

VSIPL++

```
template<class T, class U, class V> void pulseCompress(const T &in, const U &ref, const V &out) {
  int size = in.size();

  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft(in), out );
}
```

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement Support all previously stated requirements

VSIPL

Single Precision

```
void pulseCompress(int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length savedSize   = vsip_cvgetlength_f
  vsip_length savedStride = vsip_cvgetstride_f(

  vsip_length size = vsip_cvgetlength_f(in) / de

  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(

  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(s
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(s

  if (forwardFft && inverseFft && tmpView1 &&
  {
    vsip_cvputlength_f(in, size);
    vsip_cvputstride_f(in, decimationFactor);

    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2
    vsip_ccfftop_f(inverseFft, tmpView2, ou

    vsip_cvputlength_f(in, savedSize);
    vsip_cvputstride_f(in, savedStride);
  }

  if (tmpView1)   vsip_cvalldestroy_f(tmpV
  if (tmpView2)   vsip_cvalldestroy_f(tmpV
  if (forwardFft) vsip_fft_destroy_f(forwardF
  if (inverseFft) vsip_fft_destroy_f(inverseF
}
```

Double Precision

```
void pulseCompress(int decimationFactor, vsip_cvview_d *in, vsip_cvview_d *ref, vsip_cvview_d *out) {
  vsip_length savedSize   = vsip_cvgetlength_d(in);
  vsip_length savedStride = vsip_cvgetstride_d(in);

  vsip_length size = vsip_cvgetlength_d(in) / decimationFactor;

  vsip_fft_d *forwardFft = vsip_ccfftop_create_d(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_d *inverseFft = vsip_ccfftop_create_d(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);

  vsip_cvview_d *tmpView1 = vsip_cvcreate_d(size, VSIP_MEM_NONE);
  vsip_cvview_d *tmpView2 = vsip_cvcreate_d(size, VSIP_MEM_NONE);

  if (forwardFft && inverseFft && tmpView1 && tmpView2)
  {
    vsip_cvputlength_d(in, size);
    vsip_cvputstride_d(in, decimationFactor);

    vsip_ccfftop_d(forwardFft, in, tmpView1);
    vsip_cvmul_d(tmpView1, ref, tmpView2);
    vsip_ccfftop_d(inverseFft, tmpView2, out);

    vsip_cvputlength_d(in, savedSize);
    vsip_cvputstride_d(in, savedStride);
  }

  if (tmpView1)   vsip_cvalldestroy_d(tmpView1);
  if (tmpView2)   vsip_cvalldestroy_d(tmpView2);
  if (forwardFft) vsip_fft_destroy_d(forwardFft);
  if (inverseFft) vsip_fft_destroy_d(inverseFft);
}
```

Observations

- **Final SLOC count**
  - **VSIPL++ -- 6 SLOCS**
  - **VSIPL -- 40 SLOCS (20 each for double and single precision versions)**

VSIPL++

```
template<class T, class U, class V> void pulseCompress(int decimationFactor, const T &in, const U &ref, const V &out) {
  int size = in.size() / decimationFactor;

  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```


HPEC SI — Productivity (3x) — Performance (1.5x)

Algorithm Case Study

LOCKHEED MARTIN

Simple pulse compression kernel

Main Algorithm output = ifft(fft(input) * ref)

Additional requirement Port application to high performance embedded systems

VSIPL

Single Precision

```
void pulseCompress(int decimationFact
  vsip_length savedSize   = vsip_cvgetle
  vsip_length savedStride = vsip_cvgets

  vsip_length size = vsip_cvgetlength_f(i

  vsip_fft_f *forwardFft = vsip_ccfftop_cr
  vsip_fft_f *inverseFft = vsip_ccfftop_cre

  vsip_cvview_f *tmpView1 = vsip_cvcre
  vsip_cvview_f *tmpView2 = vsip_cvcre

  if (forwardFft && inverseFft && tmpVie
  {
    vsip_cvputlength_f(in, size);
    vsip_cvputstride_f(in, decimationFact

    vsip_ccfftop_f(forwardFft, in, tmpView
    vsip_cvmul_f(tmpView1, ref, tmpView
    vsip_ccfftop_f(inverseFft, tmpView2,

    vsip_cvputlength_f(in, savedSize);
    vsip_cvputstride_f(in, savedStride);
  }

  if (tmpView1)   vsip_cvalldestroy_f(tmp
  if (tmpView2)   vsip_cvalldestroy_f(tmp
  if (forwardFft) vsip_fft_destroy_f(forwar
  if (inverseFft) vsip_fft_destroy_f(invers
}
```

Observations

- **Port to embedded Mercury system**
  - **Hardware: Mercury VME chassis with PowerPC compute nodes**
  - **Software: Mercury beta release of MCOE 6.0 with linux operating system. Mercury provided us with instructions for using GNU g++ compiler**
  - **No lines of application code had to be changed**
- **Port to embedded Sky system**
  - **Hardware: Sky VME chasis with PowerPC compute nodes**
  - **Software: Sky provided us with a modified version of their standard compiler (added a GNU g++ based front-end)**
  - **No lines of application code had to be changed**
- **Future availability of C++ with support for C++ standard**
  - **Improved C++ support is in Sky and Mercury product roadmaps**
  - **Support for C++ standard appears to be improving industry wide**

VSIPL++

```
template<class T, class U, class V> void pulseCompress(int decimationFactor, const T &in, const U &ref, const V &out) {
  int size = in.size() / decimationFactor;

  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```


HPEC SI Productivity (3x) Performance (1.5x)